About Lab 7

Lab 7 is a continuation of Lab 6. In Lab 7 you will implement priority queues (via heaps) and use them along with the WebPageIndex class from Lab 6 to complete a search engine for web pages. Look out Google!

On this lab we give starter code that includes working classes for MyTreeMap, MyTreeSet, and WebPageIndex -- the three classes from Lab 6. Even if you are completely happy with your code for those, I encourage you to use the given classes until you have completely debugged your work for Lab 7 -- knowing that the trees you are working with are correct narrows down the possibilities for errors when you are working with the new code in this lab.

Note that we give you just the .class files for the Lab 6 classes; you can't modify them in any way. They will run according to the specifications given in Lab 6.

To use the given code you need to add the **csci151lab6** jar file to your build path, just as you did in Lab 6, via

Project->Properties->BuildPath->Libraries->AddJARs

If you use your own classes you should add them to the project along with the J-Soup jar file and add that jar file to your build path.

Either way you might want to test creating a WebPageIndex:

```
WebPageIndex index = new WebPageIndex( "http://www.oberlin.edu");
System.out.println(index.getWordCount());
```

If this works you have what you need to proceed.

You start coding in Lab 7 by implementing PriorityQueues. You can use an array or ArrayList to hold the tree nodes, but you will be building some rather large queues so if you use a fixed-size array you need to add into the offer( ) method the possibility of resizing the array before adding more data into it.

The methods you need to implement for myPriorityQueue<T> include

public T peek(): return the root of the binary heap

public boolean offer(T item): insert item into the heap and adjust the heap structure

public void percolateDown(int hole): If the node at index hole does not satisfy the heap property, switch its data with that of its smaller child and repeat with the child index.

public T poll(): This removes the item at the root of the heap and calls percolateDown to pass the resulting hole down through the heap.

Iterator<T> iterator(): which returns an iterator over the data in the queue. This is easy if your base type is an ArrayList.

void setComparator(Comparator<T> cmp): This installs a new comparator for the priority queue and re-heapifies the queue. This is crucial for our functionality, because every query we process will create a new comparator. If you put the root of the heap at index 1, so that the children of node i are at 2i and 2i+1, you can re-heapify with the loop

```
for (int i = size()/2; i > 0; i--)
        percolateDown(i);
```

When you are sure your priority queues are working correctly, write a comparator class for WebPageIndex objects. The provided code has a String comparator class as an example. This is only an illustration; you won't actually use the String comparator. For WebPageIndex objects, your comparator constructor will take a query string s as an argument and use it to build a comparator that gives smallest values to web pages that give the best responses to the query.

The last step is to write the ProcessQuery class. This has a main() method that opens a file from arg[0]; the contents of this file should be a list of URLs of web pages. The method builds a list of WebPageIndex objects of these pages.

It then goes into a loop that reads query strings from the console. For each query it builds a comparator for the WebPageIndex objects and turns the list of WebPageIndex objects into a MyPriorityQueue<WebPageIndex>.

Finally, it goes through a loop that repeatedly polls the priority queue (i.e, it removes the root of the heap), and prints the URL of the page that was removed. The queue is updates and the loop repeats, either until the queue is empty or until enough pages have been printed.